15-122 : Principles of Imperative Computation, Fall 2015

Written Homework 5

Due: Monday, October 5, 2015 by 6PM

Name: _____

Andrew ID: _____

Section: _____

This written homework covers big-O notation, some reasoning about searching and sorting algorithms, pointers, interfaces, and stacks and queues. You will use some of the functions from the arrayutil.c0 library, as well as the data structure interfaces introduced in lecture this week.

This is the first homework to emphasize interfaces. It's important for you to think carefully and be sure that your solutions respect the interface involved in the problem.

Print out this PDF double-sided, **staple** pages in order, and write your answers on these pages *neatly*.

The assignment is due on Monday, October 5, 2015 by 6PM.

You can hand in the assignment to your TA during lab or in the box outside of GHC 4117 (in the CS Undergraduate Program suite). WARNING: The box is removed promptly at 6PM.

You must hand in your homework yourself; do not give it to someone else to hand in.

(This page intentionally left blank. Remember to print double-sided!)

Question	Points	Score
1	5	
2	2	
3	5	
4	3	
Total:	15	

1. **Computing Overlaps**

We define the Overlap Problem as the task of computing the number of shared elements between two arrays of the same length. Assume that neither array contains duplicate entries.

Consider the following function which counts the number of integers that appear in both `A` and `B`. The code uses `linsearch`, where `linsearch(x, A, i, j)` returns the index of the first occurrence of `x` in `A[i,j)` (or -1 if `x` is not found) in linear time.

```
/*  1 */  int overlap(int[] A, int[] B, int n)
/*  2 */  //@requires 0 <= n && n <= \length(A);
/*  3 */  //@requires \length(A) == \length(B);
/*  4 */  {
/*  5 */    int count = 0;
/*  6 */    for (int i = 0; i < n; i++)
/*  7 */    //@loop_invariant 0 <= i;
/*  8 */    {
/*  9 */      if (linsearch(A[i], B, 0, n) != -1) {
/* 10 */        count = count + 1;
/* 11 */      }
/* 12 */    }
/* 13 */    return count;
/* 14 */  }
```

1pt (a) Using big-O notation, what is the worst-case runtime of this algorithm? Express your answer in its simplest, tightest form.

Solution: $O($ n^2 $)$

1pt (b) Suppose we add an additional precondition to the function:

```
//@requires is_sorted(B, 0, n);
```

With this change, explain how to modify the function to solve the Overlap Problem asymptotically faster than it currently does. (State which line(s) change and what the change(s) should be.)

Solution:

On line 9, Use binary_search instead of linear search.

1pt (c) Using big-O notation, what is the worst-case runtime complexity of your revised algorithm? Again, use the simplest, tightest form.

Solution: $O($ nlogn $)$

1pt (d) Suppose we added both of the following preconditions to the function:

```
//@requires is_sorted(A, 0, n);
//@requires is_sorted(B, 0, n);
```

With this change, describe how to adapt the merge step of Merge Sort to solve the Overlap Problem so that this new algorithm is asymptotically faster than your previous algorithm. Do NOT write code. Instead, clearly and concisely state how to modify the merge so that it counts the number of duplicates.

Solution:

1pt (e) Using big-O notation, what is the worst-case runtime complexity of this final algorithm? Again, use the simplest, tightest form.

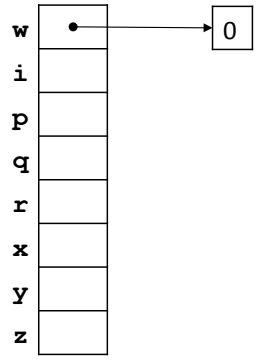
Solution: $O($ $)$

2. **Pointer Illustration**

2pts (a) Clearly and carefully illustrate the contents of memory after the following code runs. We've drawn the contents of `w`, a pointer that points into allocated memory where the number 0 is stored.

```
int* w = alloc(int);
int i = 12;
int* p = alloc(int);
int* q = p;
int* r = alloc(int);
int** x = alloc(int*);
int** y = alloc(int*);
int** z = y;
*r = i + 4;
*y = q;
**z = 4;
*x = r;
q = NULL;
i = *p + **y;
```

Solution:



3. **Implementing an Image Type Using a Struct**

In a previous programming assignment, we worked with one-dimensional arrays that represented two-dimensional images. Suppose we want to create a data type for an image along with an interface that specifies functions to allow us to get a pixel of the image or set a pixel of the image.

(You are allowed to assume that `p1 == p2` is an acceptable way of comparing pixels for equality.)

2pts (a) Complete the interface for the `image` type. Add appropriate preconditions and postconditions for each image operation. The first two functions should have at least one meaningful postcondition, but you don't have to give every conceivable postcondition.

```
// typedef __* image_t;
typedef struct image_header* image_t;

__________ image_getwidth(_____________________________________)



__________ image_getheight(_____________________________________)



__________ image_getpixel(image_t IMG, int row, int col)



__________ image_setpixel(image_t IMG, int row, int col, pixel P)



__________ image_new(__________ width, __________ height)
```

3pts (b) In the implementation of the `image_t` type, we have the following type definitions:

```
struct image_header {
    int width;
    int height;
    pixel[] data;
};
typedef struct image_header image;
```

And the following data structure invariant:

```
bool is_image(image* IMG) {
    return IMG != NULL
        && IMG->width > 0
        && IMG->height > 0
        && IMG->width <= int_max() / IMG->height
        && is_arr_expected_length(IMG->data, IMG->width * IMG->height);
}
```

The client does not need to know about this function, since it is the job of the implementation to preserve the validity of the image data structure. But the implementation must use this specification function to assure that the image is valid before and after any image operation.

Write an implementation for `image_getpixel`, assuming pixels are stored the same way they were stored in the programming assignment. Include any necessary preconditions and postconditions for the implementation.

Write an implementation for `image_new`. Include any necessary preconditions and postconditions for the implementation.

4. **Stacks, queues, and interfaces:**

2pts (a) Consider the following interface for `stack_t` that stores elements of the type `string`:

```
/* Stack Interface */
typedef ______* stack_t;

bool stack_empty(stack_t S) /* O(1), check if stack empty */
/*@requires S != NULL; @*/;

stack_t stack_new() /* O(1), create new empty stack */
/*@ensures \result != NULL; @*/
/*@ensures stack_empty(\result); @*/;

void push(stack_t S, string x) /* O(1), add item on top of stack */
/*@requires S != NULL; @*/;

string pop(stack_t S) /* O(1), remove item from top */
/*@requires S != NULL; @*/
/*@requires !stack_empty(S); @*/
```

Write a client function `stack_bottom(stack_t S)` that returns, but does not remove, the bottom element of the given stack, assuming the stack is not empty. For this question, use only the interface since, as a client, you do not know how this data structure is implemented. Do not use any stack functions that are not in the interface (including specification functions like `is_stack` since these belong to the implementation).

Solution:

```
string stack_bottom(stack_t S)
//@requires S != NULL;
//@requires !stack_empty(S);
{









}
```

1pt (b) Below is the queue interface from lecture.

```
typedef ______* queue_t;

bool queue_empty(queue_t Q) /* O(1) */
/*@requires Q != NULL; @*/;

queue_t queue_new() /* O(1) */
/*@ensures \result != NULL; @*/
/*@ensures queue_empty(\result); @*/;

void enq(queue_t Q, string e) /* O(1) */
/*@requires Q != NULL; @*/;

string deq(queue_t Q) /* O(1) */
/*@requires Q != NULL; @*/
/*@requires !queue_empty(Q); @*/ ;
```

The following is a client function `queue_size` that is intended to compute the size of the queue while leaving the queue unchanged.

```
int queue_size(queue_t Q)
//@requires Q != NULL;
//@ensures \result >= 0;
{
  int size = 0;
  queue_t C = Q;
  while (!queue_empty(Q)) {
    enq(C, deq(Q));
    size++;
  }
  while (!queue_empty(C)) enq(Q, deq(C));
  return size;
}
```

Explain why the function `queue_size` does not work and give a corrected version below:

Solution: (Explanation)

Solution: (Correct code)

```
int queue_size(queue_t Q)
//@requires Q != NULL;
//@ensures \result >= 0;
{









}
```